Зміст

[Python Core 2.0 ֍ 1) Вступ до Python 1](#_Toc192687591)

[Синтаксис, змінні, типи даних, оператори 1](#_Toc192687592)

[Числа 1](#_Toc192687593)

[Логічний тип 1](#_Toc192687594)

[Рядки 1](#_Toc192687595)

[Тип None 2](#_Toc192687596)

[Порядок виконання операцій у виразі 3](#_Toc192687597)

[Коментарі 3](#_Toc192687598)

[Вивід даних 3](#_Toc192687599)

[Введення даних 3](#_Toc192687600)

[Робота з рядками та колекції 4](#_Toc192687601)

[Колекції 4](#_Toc192687602)

[Списки 4](#_Toc192687603)

[Словники 6](#_Toc192687604)

[Методи словників 7](#_Toc192687605)

[Множини 7](#_Toc192687606)

[Математичні операції над множинами 8](#_Toc192687607)

[Заморожені множини 8](#_Toc192687608)

[Кортежі 9](#_Toc192687609)

[Методи рядків 9](#_Toc192687610)

[Зрізи у Python (Slice) 10](#_Toc192687611)

[Python Core 2.0 ֍ 2) Контроль потоку та функцій 11](#_Toc192687612)

[Умовні оператори, цикли 11](#_Toc192687613)

[Умовне виконання коду 11](#_Toc192687614)

[Логічні вирази 11](#_Toc192687615)

[Оператор is 11](#_Toc192687616)

[Булева алгебра 12](#_Toc192687617)

[Оператор and (і) 12](#_Toc192687618)

[Оператор or (або) 12](#_Toc192687619)

[Оператор not (ні) 13](#_Toc192687620)

[Тернарні операції 13](#_Toc192687621)

[Оператор match 13](#_Toc192687622)

[Цикли 14](#_Toc192687623)

[Розширення можливостей циклу for 15](#_Toc192687624)

[Функція Range 15](#_Toc192687625)

[Функція Zip 16](#_Toc192687626)

[Цикли та словники 16](#_Toc192687627)

[Винятки 17](#_Toc192687628)

[Механізм обробки винятків 17](#_Toc192687629)

[Функції, область видимості змінних (LEGB) 18](#_Toc192687630)

[Робота з функціями 18](#_Toc192687631)

[Області видимості (LEGB) 19](#_Toc192687632)

[Ключові аргументи функції 20](#_Toc192687633)

[Змінна кількість параметрів 20](#_Toc192687634)

[Рекурсія 22](#_Toc192687635)

[Стек викликів рекурсії 22](#_Toc192687636)

[Python Core 2.0 ֍ 3) Робота з датою, часом та розширена робота з рядками 24](#_Toc192687637)

[Модулі datetime та time. Робота з випадковими величинами. Модуль math. 24](#_Toc192687638)

[Робота з датою 24](#_Toc192687639)

[Робота з ISO форматом дати 27](#_Toc192687640)

[Робота з часовими зонами 28](#_Toc192687641)

[Робота з часом 28](#_Toc192687642)

[Робота з випадковими величинами 29](#_Toc192687643)

[Модуль math 30](#_Toc192687644)

[Регулярні вирази та розширена робота з рядками 31](#_Toc192687645)

[Просунута робота з рядками 31](#_Toc192687646)

[Спеціальні символи 31](#_Toc192687647)

[Методи рядків 32](#_Toc192687648)

[Translate 33](#_Toc192687649)

[Форматування рядків 34](#_Toc192687650)

[Регулярні вирази 35](#_Toc192687651)

[Основні функції модуля re: 36](#_Toc192687652)

[Python Core 2.0 ֍ 4)Робота з файлами та модульна система 38](#_Toc192687653)

[Робота з файлами та обробка винятків 38](#_Toc192687654)

[Обробка винятків. Менеджер контексту 39](#_Toc192687655)

[Робота з нетекстовими файлами у Python 39](#_Toc192687656)

[Перетворення чисел у байт-рядки 40](#_Toc192687657)

[Кодування рядків (ASCII, UTF-8, CP1251) 40](#_Toc192687658)

[Масив байтів 41](#_Toc192687659)

[Порівняння рядків 41](#_Toc192687660)

[Робота з архівами 42](#_Toc192687661)

[Основи модуля pathlib 43](#_Toc192687662)

[Робота з модулями та створення віртуального оточення 46](#_Toc192687663)

[Імпорт пакетів та модулів 46](#_Toc192687664)

[Функція dir() 46](#_Toc192687665)

[Службова змінна \_\_name\_\_ 47](#_Toc192687666)

[Модуль sys. Обробка аргументів командного рядка 47](#_Toc192687667)

[Файл \_\_init.py\_\_ 48](#_Toc192687668)

[Пакетний менеджер pip та створення віртуального оточення 49](#_Toc192687669)

[Створення віртуального оточення 50](#_Toc192687670)

[Бібліотека colorama 51](#_Toc192687671)

[Єдина точка входу проєкту 51](#_Toc192687672)

[Python Core 2.0 ֍ 5) Функціональне програмування та вбудовані модулі Python 52](#_Toc192687673)

[Python Core 2.0 ֍ 6) Основна робота з класами 52](#_Toc192687674)

[Python Core 2.0 ֍ 7) Розширене Об'єктно-Орієнтоване Програмування в Python 52](#_Toc192687675)

[Python Core 2.0 ֍ 8) Серіалізація та копіювання об'єктів в Python 52](#_Toc192687676)

Python Core 2.0 ֍ 1) Вступ до Python

Синтаксис, змінні, типи даних, оператори

**Елементи будь-якої програми:**

- Введення (отримання даних)

- Перетворення (обробка даних)

- Виведення (виведення даних)

**Типи даних у Python:**

1. Числові типи
2. Логічний тип (Boolean).
3. Текстовий тип або рядки.
4. Тип None

Числа

* Цілі числа мають позначення як int.
* Дійсні числа(десяткові дроби) позначаються як **float.** Точність **float** обмежена, що може призвести до певних помилок округлення у складних обчисленнях. a = 0.2 + 0.1 # 0.30000000000000004. Якщо при обчисленнях точність стає важливою, програмісти на Python використовують спеціальний модуль [Decimal](https://docs.python.org/3/library/decimal.html).
* Комплексні числа позначаються як **complex.** (**a + bj**, де **a** є реальною частиною, а **b** — уявною частиною, і **j** є символом уявної одиниці.)

Логічний тип

Логічний (булевий) тип у Python, відомий як **bool**, є основним типом даних, який використовується для представлення істинності або хибності. (**True,** **False**)

Рядки

Рядкові змінні — це впорядковані незмінні набори символів.

Тип рядка у Python, зазначений як **str**. Для того щоб Python зрозумів, що ми хочемо створити рядкову змінну, необхідно взяти символи рядка в лапки. (одинарних **'...'**, подвійних **"..."** або потрійних лапок — **'''...'''** чи **"""...""".** Потрійні лапки використовуються для багаторядкових рядків).

Основна операція, яка реалізована для рядків — це об'єднання рядків ([конкатенація](https://uk.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%BA%D0%B0%D1%82%D0%B5%D0%BD%D0%B0%D1%86%D1%96%D1%8F)). Конкатенація рядків реалізована з використанням оператора додавання +.

s1 = "Hello"

s2 = "world!"

joined\_string = s1 + " " + s2

Для зручності виведення тексту у Python застосовують спеціальну конструкцію f-рядка.

f-рядок — це такий шаблон, який дозволяє зручним чином генерувати рядок, підставляючи результат виконання виразів у потрібне місце в шаблоні в фігурних дужках {}.

В f рядку **{змінна:.2f}**. Вираз **.2** це кількість знаків після десяткової крапки. Символ **f** означає, що число має бути відображене у форматі дійсного числа.

Попередній приклад ми могли б оформити гарніше за допомогою f-рядків.

s1 = 'Hello'

s2 = 'world!'

joined\_string = f"{s1} {s2}" *# Hello world!*

Тип None

Тип **None** у Python використовується для позначення відсутності значення або "нічого". Найпоширеніше застосування **None** для ініціалізації змінної — коли реальне значення має бути надано пізніше.

Арифметичні оператори:

* + **Додавання**
* - **Віднімання**
* \* **Множення**
* / **Ділення** (завжди повертає результат ділення як дійсне число float)
* % **Залишок від ділення**
* // **Ділення без остачі** (округлення вниз до найближчого цілого числа. Якщо обидва числа — int, результат також буде int. Якщо хоча б один з чисел float, результат також буде float.
* \*\* **Піднесення до степеня**

Порядок виконання операцій у виразі

1. **Дужки ()**
2. **Піднесення до степеня \*\***
3. **Унарні плюс + та мінус –**
4. **Множення \*, ділення /, цілочисельне ділення //, остача від ділення %**:
5. **Додавання + та віднімання -.**
6. **Операції порівняння ==, !=, >, <, >=, <=**
7. **Логічні операції not, and, or**: В кінці виконуються логічні операції, де **not** має вищий пріоритет, ніж **and**, який, у свою чергу, має вищий пріоритет, ніж **or**.

Коментарі

При виконанні програми комп'ютер ігнорує всі коментарі в коді, вони є там виключно для людини. Для Python коментар починається із символу ґрат #.

Повний перелік вбудованих функцій можна знайти на [**сторінці офіційної документації**](https://docs.python.org/3/library/functions.html).

Вивід даних

Функція **print()**застосовується для виведення інформації на стандартний вихід, який зазвичай є консоллю або терміналом.

Можна виводити кілька елементів одночасно, відокремлюючи їх комами.

**print("Привіт", end=" ")** не перейде на новий рядок після виведення.

Введення даних

Для введення даних з консолі використовується функція input.

Функція **input()** може приймати рядок як параметр, який буде виведений на екран, перш ніж користувач зробить введення.

a = input("Рядок:") *# На екрані буде: Рядок:*

**Приведення типів**

Функція input повертає тип str але можна перетворити тип в int:

age = int(input("How old are you? "))

Для перетворення рядків у числа з крапкою можна використовувати функцію float, у рядок функцією str, до булевого функцією bool (**0**, порожні колекції (**[]**, **()**, **{}**) та **None** перетворюються на **False**. Усе інше — на **True**).

Cимвол **\** використовується в кінці рядка для вказівки на те, що вираз продовжується на наступному рядку. Після **\** не повинно бути жодних інших символів, навіть пробілів або коментарів, до кінця рядка.

Робота з рядками та колекції

Колекції

Колекція у Python — змінна-контейнер, що зберігає набір значень одного або різних типів.

Основні властивості колекцій:

* *Впорядкованість*. можна звертатися до своїх елементів за індексом (номером)
* *Змінність*. можна додавати, видаляти, заміняти елементи колекції.
* *Унікальність*.

Основні типи колекцій у Python:

1. **Списки (Lists).** Впорядкована, змінна колекція.
2. **Кортежі (Tuples).** Схожі на списки, але вони незмінні.
3. **Словники (Dictionaries).** Колекції пар ключ-значення (кожний ключ унікальний).
4. **Множини (Sets).** Неупорядковані колекції унікальних елементів.
5. **Заморожені множини** (**Frozen Sets).** Це незмінні версії множин.

Списки

Щоб створити порожній список:

my\_list = list()

empty\_list = []

Ви можете створити список, просто перерахувавши його елементи у квадратних дужках, розділених комами.

Списки можуть містити різні типи даних.

**.append(…)** - додати щось в кінець списку,

**.remove(…)** - приймає елемент списку, який потрібно видалити.

Доступу за індексом виглядає так:

some\_iterable = ["a", "b", "c"]

first\_letter = some\_iterable[0] #a

Python підтримує індексування елементів з кінця. Якщо додати - і вказати номер елемента з кінця (перший елемент з кінця — це -1,).

some\_iterable = ["a", "b", "c"]

letter = some\_iterable[-1] #с

Змінимо елемент списку на …:

список[елемент] = …

**.pop(i)** - повернути елемент за індексом i та видалити його зі списку. За замовчуванням i = -1, і метод повертає останній елемент списку.

елемент = список.pop(і)

список\_1.extend(список\_2) – додаємо 2с. до 1 списку

**.insert(i, x)** - Вставка елемент x на позицію з індексом i у список

**.clear(…)** - Очищення списку від елементів.

**.index(…)** - знаходження індексу першого входження елемента у списку. Зберігає індекс в змінній.

Індекс = список**.index(**елемент**)**

**.count(…)** - підрахунок кількості разів, скільки певний елемент зустрічається у списку.

✂️ Цей код можна запустити!

my\_list = [1, 2, 3, 4, 2, 2, 5, 2]

count\_2 = my\_list.count(2)

print(count\_2) # Виведе 4, оскільки число 2 зустрічається 4 рази

**len(…)** - кількість елементів у колекції.

print(len(my\_list))

**.sort()** - сортування елементів списку в порядку зростання або спадання(за замовчуванням від меншого до більшого). Змінює сам список. В порядку спадання **reverse=True.**

список.sort()

список.sort(reverse=True)

список.sort(key=len)- сортувати за ключем (довжина сл., спочатку короткі)

**sorted()** –як sort(), але повертає новий відсортований список (будб-яку колекцію).

sorted\_nums = sorted(nums)

sorted\_nums\_desc = sorted(nums, reverse=True)

sorted\_words = sorted(words, key=len)

список\_copy = список.copy() - повернути копію списку

**.reverse()** – змінює порядок елементів у списку на зворотний.

Словники

Словник — це контейнер, який зберігає пари ключ-значення. Ключ (унікальний) – незмінний тип даних.

Створення пустого словника:

my\_dict = {}

Cтворення заповненого словника:

my\_dict = {"name": "Alice", "age": 25, "city": "New York"}

Для доступу до значення у словнику за ключем:

my\_dict = {"name": "Alice", "age": 25, "city": "New York"}

print(my\_dict["name"]) # Виведе 'Alice'

Змінити значення або додати нову пару ключ-значення, просто використовуйте ключ для присвоєння нового значення:

my\_dict["age"] = 26 *# Змінює вік на 26*

my\_dict["email"] = "alice@example.com" *# Додає нову пару*

**del** - видалити елемент за ключем:

del my\_dict["age"]

Для перевірки, чи є ключ у словнику, використовуй оператор **in**:

✂️ Цей код можна запустити!

print("name" **in** my\_dict) *#True*

Методи словників

* **Метод .pop()** – дозволяє видалити елемент за вказаним ключем і повернути його значення. (**age = my\_dict.pop("age")**)
* **Метод .update({…; …})** – використовується для оновлення словника іншим словником або парами ключ-значення.
* **Метод clear()** – очищує словник, видаляючи всі його елементи.
* **Метод copy()** – створює копію словника (**new\_dict = my\_dict.copy()**)
* **Метод get()** – отримання значення за ключем зі словника (не викликає помилку, якщо ключ не знайдено, повертає **None**)

age = my\_dict.get("age") *# Поверне число*

gender = my\_dict.get("gender") *# Поверне None, оскільки "gender" немає*

Множини

Множини — це неврегульований контейнер, який містить тільки унікальні елементи. У множину можна додавати тільки незмінні типи даних.

Щоб створити порожню множину:

empty\_set = set()

Множини не мають визначеного порядку елементів. Це означає, що ви не можете звернутися до елементів множини через індекс або отримати їх у певному порядку.

Для створення заповненої множини:

a = set('hello') *#4 елемента h, e, l, o*

Або:

b = {1, 2, 3, 4, 5}

Унікальність множини передбачає, що якщо множина вже містить такий елемент, то спроба додати ще один такий самий нічого не змінить.

numbers = {1, 2, 3, 1, 2, 3}

Множини підтримують наступні методи:

* **.add(elem)** — додає елемент у множину
* **.remove(elem)** — видаляє елемент із множини, викликає виняток, якщо такого елемента немає
* **.discard(elem)** — видаляє елемент із множини і не викликає виняток, якщо його немає

Математичні операції над множинами

a = {1, 2, 3}

b = {3, 4, 5}

**Перетин** - елементи, які є в обох множинах. ( & або метод .intersection ):

print(a.intersection(b)) # {3}

print(a & b) # {3}

**Різниця** елементи, які містяться в першій множині, але не містяться в другій. ( **-** або метод difference ):

print(a.difference(b)) # {1, 2}

print(a - b) # {1, 2}

**Симетрична різниця** всі елементи, окрім тих які містяться в двох множинах. ( ^ або метод symmetric\_difference ):

print(a.symmetric\_difference(b)) # {1, 2, 4, 5}

print(a ^ b) # {1, 2, 4, 5}

**Об'єднання** всі елементи з обох множин, але без дублікатів. ( | або методу union):

print(a.union(b)) # {1, 2, 3, 4, 5}

print(a | b) # {1, 2, 3, 4, 5}

Заморожені множини

Заморожену множину можна створити:

my\_frozenset = frozenset([1, 2, 3, 4, 5])

Неможливо змінити елементи замороженої множини після її створення.

Операції, над замороженими множинами:

union = a | b # Об'єднання множин

intersection = a & b # Перетин множин

difference = a - b # Різниця множин

symmetric\_difference = a ^ b # Симетрична різниця

У цих прикладах результатом кожної операції буде нова заморожена множина.

Кортежі

Кортежі в Python — структура даних, подібна до списків, але **вони незмінні**.

Створити порожній кортеж:

my\_tuple = tuple() *# або*

my\_tuple = ()

Створення непорожнього кортежу:

my\_tuple = (1, 2, 3)

my\_tuple = (1,) *# кома обов’язкова*

Кортеж, як і список, може містити різні типи даних:

my\_tuple = (1, "Hello", 3.14)

Кортеж можна створити без дужок, ця операція називається упакування кортежу:

my\_tuple = 1, "Hello", 3.14

Отримати доступ до елементів:

first\_item = my\_tuple[0] *# Отримати перший елемент*

Методи рядків

**Рядок** — це незмінна впорядкована послідовність символів у деякому кодуванні. (За замовчуванням використовується кодування [UTF-8](https://uk.wikipedia.org/wiki/UTF-8)) Кортеж.

**Впорядкована послідовність** означає, що до елементів рядка можна звертатися за індексом:

s = "Hello world!"

print(s[0])# H

print(s[-1])# !

**Незмінна послідовність** означає, що змінити його не можна.

s = "Hello world!"

s[0] = "Q"*# Тут буде викликано виняток (помилка) TypeError*

текст.upper() # Верхній регістр

текст.lower() # Нижній регістр

текст.startswith("…") # перевірити, що рядок починається з "…"

текст.endswith("…") # перевірити, що рядок закінчується на "…"

s = "hello world".capitalize() *# Результат: "Hello world"* речення з великої

s = "hello world".title() *# Результат: "Hello World"* всі сл. з великої

"123".isdigit() # True (рядок складається тільки з цифр)

"hello".isalpha() # True (рядок складається тільки з літер)

" ".isspace() # True (рядок складається тільки з пробілів)

**Форматування рядків**

"{}…{}".format(змінна1, змінна2 …) - форматування рядків

Зрізи у Python (Slice)

Ось основний синтаксис:

послідовність[початок : кінець : крок]

Ми можемо використовувати від'ємні індекси у зрізах.

reverse\_numbers = numbers[::-1]

copy\_numbers = numbers[:] *# копіювати кортеж*

Зріз **[:]** вказує на вибір усіх елементів зі списку, починаючи з першого і закінчуючи останнім.

Python Core 2.0 ֍ 2) Контроль потоку та функцій

Умовні оператори, цикли

Умовне виконання коду

Умовний оператор у Python має такий синтаксис:

if <умова1>: *# повертає True або False*

<тіло if-блоку> *# виконуються, якщо умова є True*

elif <умова2>:

<тіло elif-блоку> *# виконуються, якщо умова1 є False*

else:

<тіло else-блоку> *# виконуються, якщо всі умови є False*

Під час виконання умовного оператора інтерпретатор Python перевіряє умови зверху вниз, доки не знайде те, яке виконується, потім виконає вираз для цієї умови та вийде з перевірки умов.

Логічні вирази

У мовний оператор if ... elif ... else може приймати змінні типу bool або будь-який вираз, який він виконає і результат перетворить в bool. Для зручності у Python є механізм неявного приведення будь-якого типу до типу bool. Правила приведення до bool достатньо інтуїтивні.

*Правило перше* - число 0 приводиться до False

*Правило друге* - значення None приводиться до False.

*Правило третє* - порожній контейнер, рядок тощо, приводиться до False.

*Правило останнє* - все інше приводиться до True

Оператор is

Оператор **is** у Python використовується для перевірки того, чи два об'єкти вказують на одну і ту ж область пам'яті, тобто чи вони є одним і тим же об'єктом.

Цей оператор відрізняється від оператора **==**, який перевіряє рівність значень об'єктів.

a = [1, 2, 3]

b = a

c = [1, 2, 3]

print(a is b) # True

print(a is c) # False

Однак основне його застосування - це перевірка, чи змінна є **None**.

if змінна is None:

# Робимо щось, якщо 'змінна' є 'None'

Булева алгебра

[**Булева алгебра**](https://uk.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%B5%D0%B1%D1%80%D0%B0_%D0%BB%D0%BE%D0%B3%D1%96%D0%BA%D0%B8) — вивченням істинності виразів та їх обробкою.

У найпростішій формі булева алгебра оперує з двома значеннями: **True** (істина) і **False** (неправда).

Основні операції в булевій алгебрі включають:

* **AND** (і): Операція повертає **True**, якщо обидва операнди є **True**. Наприклад, **True AND True** є **True**, в той час як **True AND False** є **False**.
* **OR** (або): Операція повертає **True**, якщо хоча б один з операндів є **True**. Наприклад, **True OR False** є **True**.
* **NOT** (ні): Унарна операція, яка інвертує значення; **True** стає **False**, а **False** стає **True**.

Оператор and (і)

Таблиця істинності для **and**:

|  |  |  |
| --- | --- | --- |
| A | B | A and B |
| True | True | True |
| True | False | False |
| False | True | False |
| False | False | False |

Оператор or (або)

Таблиця істинності для **or**:

|  |  |  |
| --- | --- | --- |
| A | B | A or B |
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

Оператор not (ні)

Таблиця істинності для **not**:

|  |  |
| --- | --- |
| A | not A |
| True | False |
| False | True |

На останок розглянемо завдання "FizzBuzz”. Завдання часто зустрічається на співбесідах з програмування. Його суть полягає в написанні програми, яка для заданого числа:

**Блоки інструкцій та тернарні оператори**

Для виділення одного рівня вкладеності для блоку інструкцій використовувати **4 пробіли**. Ви можете використати символи табуляції для виділення блоку інструкцій, це не помилка, але такий спосіб не рекомендується.

Тернарні операції

Тернарні оператори в Python є елегантним способом вираження умовних виразів у скороченій формі.

Вираз змінна = "тіло if-блоку" if умова else "тіло if-блоку" використовує тернарний оператор для присвоєння рядка "тіло if-блоку" змінній, якщо умова є істинним True, і рядка "тіло if-блоку", якщо умова є неправдивим False

Ще одним корисним варіантом тернарного оператора в Python є використання оператора **or** для швидкого визначення значення.

some\_data = None

msg = some\_data or "Не було повернено даних"

Якщо **some\_data** є **None** або неправдивим(False), **msg** отримає значення **"Не було повернено даних"**.

Оператор match

Оператор **match** дозволяє порівнювати значення з декількома шаблонами і виконувати різні блоки коду в залежності від того, який шаблон відповідає значенню.

Структура оператора match

match змінна:

case шаблон1:

# виконати код для шаблону 1

case шаблон2:

# виконати код для шаблону 2

case \_:

# виконати код, якщо не знайдено відповідностей

Оператор **match** - це свого роду розширена і більш гнучка версія оператора **if-elif-else**. Він дозволяє порівнювати значення з рядом шаблонів і, залежно від відповідності, виконувати певні дії.

Але оператор має більш розширену сферу використання.….

Цикли

Існують два основних типи циклів: **for** та **while**.:

* Цикл **for** в Python використовується для ітерації по елементах будь-якої послідовності (наприклад, списку, кортежу, рядка) або інших ітерованих об'єктів;

for element in sequence:

*# виконувати дії з element*

* Цикл **while** виконує блок коду, поки задана умова є істинною (**True**). Як тільки умова стає неправдивою (**False**), цикл закінчується.

while condition:

# виконувати дії, поки condition є True

**Цикл for**

У Python цикл for використовується для перебору усіх елементів контейнерів або ітерованих об'єктів, наприклад, списків. Інструкції, які знаходяться у тілі циклу, будуть виконані стільки разів, скільки елементів у колекції.

for змінна in контейнер:

виконувати дії з змінною

**Цикл while**

Цикл **while** продовжує виконувати блок інструкцій, поки задана умова залишається істинною(**True**). Цикл **while** часто використовується, коли кількість ітерацій заздалегідь невідома або коли ітерації залежать від змінних, які можуть бути модифіковані в процесі виконання циклу.

Синтаксис циклу while наступний:

while умова:

# Блок коду для виконання

При використанні циклів **while** важливо забезпечити, що умова циклу змінюватиметься в процесі його виконання, щоб уникнути нескінченних циклів.

**«Нескінченні цикли» та break**

Бувають ситуації, коли необхідно вийти з циклу до завершення ітерації, не дочекавшись, доки станеться чергова перевірка умови. Для цього є команда break. Команда break зупиняє цикл в момент виклику і не завершує ітерацію.

while True:

break

**Завершення ітерації за допомогою continue**

Також для того, аби одразу перейти до наступної ітерації циклу без виконання виразів, що залишилися, є команда continue. Виклик цієї команди у тілі циклу призводить до того, що вирази цієї ітерації, що залишилися, не будуть виконані, а інтерпретатор одразу перейде до наступної ітерації або перевірки умови.

Оператори continue та break працюють тільки всередині одного циклу. В ситуації вкладених циклів немає способу вийти з усіх циклів одразу.

Також використання continue або break поза циклом призводить до синтаксичної помилки.

Розширення можливостей циклу for

Функція Range

Функція **range** у поєднанні з циклом **for** у Python є потужним інструментом для контролю повторення дій певну кількість разів.

Функція **range** створює послідовність чисел. Вона може бути використана різними способами:

* **range(stop)**: Створює послідовність чисел від 0 до **stop - 1**.
* **range(start, stop)**: Генерує числа від **start** до **stop - 1**.
* **range(start, stop, step)**: Створює числа від **start** до **stop - 1**, з кроком **step**.

**Функція Enumerate**

Функція **enumerate** використовується для одночасного отримання індексу(від 1) та значення елементів ітерованого об'єкта. Це корисно, коли вам потрібно отримати доступ до індексу елементів під час ітерації.

for index, value in enumerate(список\_з\_value):

print(index, value)

Функція Zip

Функція **zip** використовується для ітерації по декількох ітерованих об'єктах одночасно. **zip(list1, list2)** об'єднує елементи з **list1** та **list2**, і цикл **for** проходить по створеним словосполученням.

for number, letter in zip(list1, list2):

print(number, letter)

Коли колекції, передані в **zip**, мають різну довжину, **zip** обробляє елементи до тих пір, поки не закінчаться елементи в найкоротшій колекції.

Цикли та словники

Ітеруючи за словником, ви перебираєте ключі словника. Таку саму поведінку можна отримати, використовуючи метод keys, що хочете перебрати ключі:

for key in numbers.keys():

print(key)

Щоб перебрати значення словника скористаємося методом values:

for val in numbers.values():

print(val)

Щоб перебрати пари ключ значення словника треба використати метод items. На кожній ітерації ми отримаємо пару (ключ, значення):

for key, value in numbers.items():

print(key, value)

Важливо пам'ятати, що **не можна** робити, поки ітеруєтеся за словником\списком: не можна видаляти елементи, не можна додавати елементи. Але можна перезаписувати значення, якщо ви ітеруєтеся за ключами.

Винятки

Виключення у Python — це помилка на рівні інтерпретатора, викликана неможливістю виконати той або інший оператор з будь-яких причин (змінна не існує, синтаксична помилка, відсутній атрибут, операція ділення на нуль тощо).

**Основні типи винятків у Python**

SyntaxError — синтаксична помилка.

IndentationError — помилка, яка виникає, якщо у виділенні блоків інструкцій пробілами допущена помилка.

TabError виникає, якщо в одному файлі використовувати пробіли і табуляції для виділення блоків інструкцій.

TypeError виникає, коли операція зі змінною цього типу неможлива. # 2 / *'a'*

ValueError виникає, коли тип операнда відповідний, але значення таке, що операцію неможливо виконати. # int("a")

ZeroDivisionError — ділення на нуль.

Механізм обробки винятків

Для обробки винятків існує оператор

try:

блок коду

except тип вийнятку: *# можна додати декілька блоків except*

те, що треба виконати

else: *# виконається, тільки якщо винятків не сталося. (НЕ ОБОВ’ЯЗКОВО)*

те, що треба виконати

finally: *# виконається у будь-якому разі. (НЕ ОБОВ’ЯЗКОВО)*

те, що треба виконати

Наївним розв'язанням цієї проблеми буде повсюдне використання перевірок if на коректність введеного користувачем або іншим застосунком значення.

Функції, область видимості змінних (LEGB)

Робота з функціями

Функції – це спосіб групування коду, який виконує певну задачу.

**Створення та виклик функцій**

Функція оголошується за допомогою ключового слова def. Після ключового слова вказується ім'я функції, за яким йде пара дужок (), у яких можна вказати імена деяких змінних, та двокрапка : в кінці рядка. Далі слідує блок команд, що складають функцію.

**Аргументи функції**

Параметри функції – це деякі вхідні дані, які ми можемо передати функції, щоб отримати результат, що відповідає цим даним. Параметри вказуються в дужках при оголошенні функції та розділяються комами. Аналогічно ми передаємо значення, коли викликаємо функцію.

**Типізація параметрів функції**

Вказівки очікуваних типів параметрів.

def функція(a: int, b: int):….

**Повернення результату**

Повернення результату з функції виконується за допомогою оператора **return**. Ви можете явно вказати тип даних, який повертається функцією, використовуючи анотації типів.

Базовий синтаксис для повернення значення з функції виглядає наступним чином:

def my\_function() -> ReturnType:

*# виконати дії*

return result

**ReturnType** вказує на тип даних, який функція має повертати. (**int**, **float**, **str**, **list**, **dict**)

**Принципи змінності об'єктів у Python**

У Python всі об'єкти передаються за посиланням, але важливо розуміти різницю між змінними (mutable) та незмінними (immutable).

Незмінні типи в Python — не можуть бути змінені після їх створення(**int**, **float**, **str**, **tuple**). **Не змінює оригінальний аргумент, який передається в функцію.**

str\_var = "оригінал"

print(функція(str\_var)) *# виведе: змінено*

print(str\_var) *# виведе: оригінал*

Змінні типи, як списки **list**, словники **dict**, множини **set**, можуть змінюватися. **Зміни, зроблені всередині функції, відображаються на оригінальному об'єкті.**

my\_list = [1, 2, 3]

функція(my\_list)

print(my\_list) *# виведе: [1, 2, 3, 4]*

Використовуйте метод **copy()** для створення копій змінних об'єктів, якщо не хочете змінювати оригінал. В функції перед діями: ім’я = кортеж.copy()

✂️ Цей код можна запустити!

**ord(буква)** для отримання букви кодом ASCII

Області видимості (LEGB)

Область видимості — це область у програмі (коді), в межах якої ви можете звернутися за ім'ям до вмісту змінної. Python використовує **LEGB-правило** для пошуку змінних:

1. **L - Local (Локальна)**: змінні всередині функції (змінні ізольовані(змінні існують лише в межах блоку коду наприклад функції))

x = 50

def func() -> None:

x = 2

print('Зміна локального x на', x)

func() *# Зміна локального x на 2*

print('Глобальний x як і раніше', x) *# x як і раніше 50*

1. **E - Enclosing (Охоплювана)**: змінні у функції, що містить вкладену функцію. Внутрішня функція може читати але не змінювати змінні, визначені в функції, що її охоплює. Щоб змінювати змінні в функції, що охоплює внутр. функцію, необхідно в внутрішній ф. прописати nonlocal змінна і фикликати внутр. ф. в зовн. ф.
2. **G - Global (Глобальна)**: Це область видимості на рівні модуля або сценарію. Змінні, визначені на цьому рівні, доступні у всьому модулі.

Для того, щоб змінити глобальну змінну(в не функції) всередині функції, необхідно використовувати ключове слово **global** змінна/і.

x = 50

def func():

global x

x = 2

func() *# х тепер 2*

1. **B - Built-in (Вбудована)**: (вбудовані функції, **len**, **range** і т.д.)

Ключові аргументи функції

Ключові аргументи(параметри) в функціях — це спосіб передачі аргументів функції, при якому кожному аргументу присвоюється ім'я. Це дозволяє вказувати аргументи у будь-якому порядку під час виклику функції. Значеннями за замовчуванням можуть бути забезпечені тільки параметри, що знаходяться у кінці списку параметрів.

def функція(name, message="Привіт"): …

# **name** — це позиційний параметр, а **message** — ключовий параметр.

# передача власного значення для message

greet("Марія", message="Добрий день")

Змінна кількість параметрів

Змінна кількість параметрів у функції в Python дозволяє функції приймати нефіксовану кількість аргументів.

1. Параметр \*args. Він дозволяє функції приймати довільну кількість позиційних аргументів. Аргументи, передані функції, зберігаються у вигляді кортежу.
2. Параметр \*\*kwargs. Він дозволяє функції приймати довільну кількість ключових аргументів. Але аргументи, передані функції, зберігаються вже у вигляді словника.

Приклад використання параметру \*args

def print\_all\_args(\*args):

for arg in args:

print(arg)

print\_all\_args(1, 'hello', True) *# Виведення: 1\_hello\_True*

Приклад використання параметру \*\*kwargs

Параметр **\*\*kwargs** використовується, щоб функція приймала довільну кількість ключових аргументів.

Під час виклику функції, всі ключові аргументи, передані після останнього визначеного аргументу, будуть упаковані в словник.

def greet(\*\*kwargs):

for key, value in kwargs.items():

print(f"{key}: {value}")

greet(name="Alice", age=25)

Проте ніхто не забороняє параметри **\*args** та **\*\*kwargs** використовувати разом у функції, що дозволяє їй приймати довільну кількість як позиційних, так і ключових аргументів.

def example\_function(\*args, \*\*kwargs):

print("Позиційні аргументи:", args) *#(1, 2, 3)*

print("Ключові аргументи:", kwargs) *# {'name': 'Alice', 'age': 25}*

example\_function(1, 2, 3, name="Alice", age=25)

❗ Завжди використовуйте **\*args** перед **\*\*kwargs** у визначенні функції, оскільки це обов'язкове правило синтаксису Python.

**Розпакування списків та словників**

Оператори \* та \*\* також використовують для розпаковки списків та словників. Розпакування списку дозволяє присвоїти елементи списку окремим змінним.

my\_list = [1, 2, 3]

a, b, c = my\_list

Якщо вам потрібно ігнорувати деякі з елементів можна використовувати **\_**:

a, \_, c = my\_list

Можна також розпакувати частину списку, використовуючи **\***:

a, \*rest = my\_list *# rest стане списком = [2, 3].*

Розпакування словників часто використовується для передачі іменованих аргументів у функції (функція(\*\*ім’я\_словнка)), особливо коли кількість аргументів або їхні імена не відомі заздалегідь.

def greet(name, age):

print(f"Hello {name}, you are {age} years old.")

person\_info = {"name": "Alice", "age": 25}

greet(\*\*person\_info)

Рекурсія

**Рекурсія** — це концепція в програмуванні, коли функція викликає саму себе в рамках власного виконання.

Основні компоненти рекурсії:

1. **Базовий випадок:** Це умова, при якій рекурсія припиняє виклик самої себе, щоб уникнути нескінченного циклу.
2. **Рекурсивний випадок:** Це умова, за якої функція викликає саму себе з новими аргументами.

def factorial(n):

if n == 0: *# базовий випадок*

return 1

else:

return n \* factorial(n-1) *# рекурсивний випадок*

print(factorial(5)) *# виведе 120*

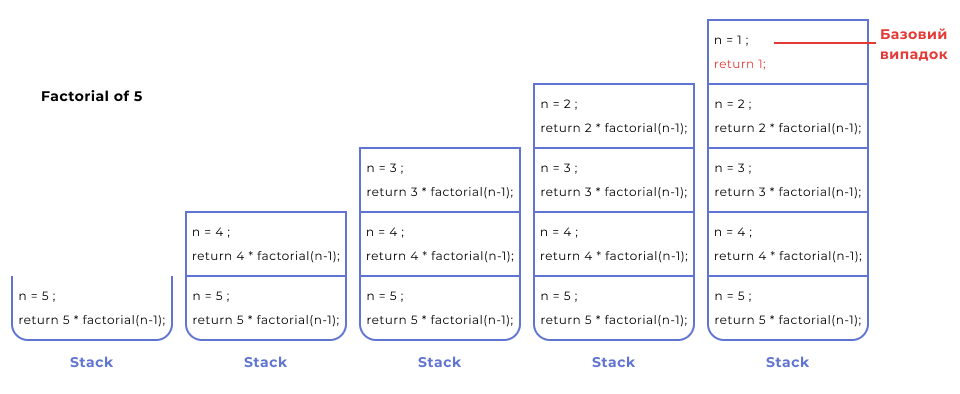
Цей код працює тому, що кожен виклик **factorial(n-1)** працює на дедалі меншому числі аж до того моменту, коли **n** дорівнює 0, і відбувається базовий випадок.

Однією з потенційних проблем при використанні рекурсії є переповнення стеку. Якщо рекурсивні виклики функції продовжуються без досягнення базового випадку, стек викликів може переповнитись, що призведе до помилки виконання.

Стек викликів рекурсії

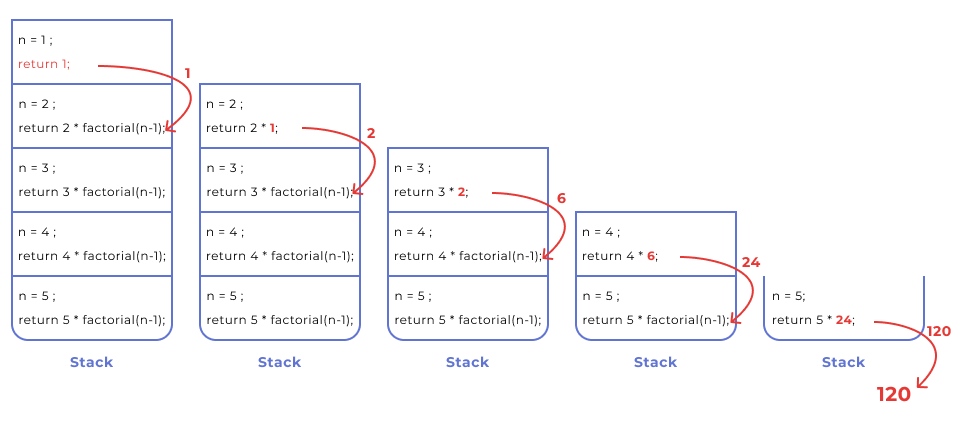
Коли функція викликає себе рекурсивно, кожен новий виклик функції зберігається в стеку викликів. Коли базовий випадок досягнуто, виклики функцій починають "розвантажуватись" зі стеку, що дозволяє розрахувати кінцевий результат.

Коли ви викликаєте **factorial(5)**, створюється перший шар у стеку викликів. Оскільки ця функція викликає себе знову, створюється ще один шар стеку для **factorial(4)**, потім для **factorial(3)** і так далі, доки не буде досягнуто базового випадку **factorial(1)**. У цей момент рекурсія досягла своєї найбільшої глибини, і стек викликів містить п'ять шарів.



Завершення виконання **factorial(1)** починає процес "розвантаження" стеку.

Кожен шар, починаючи з останнього, обробляється, і результат кожного виклику повертається до попереднього виклику, доки весь стек не буде оброблений, і ми не отримаємо кінцевий результат для **factorial(5)**.



Python Core 2.0 ֍ 3) Робота з датою, часом та розширена робота з рядками

Модулі datetime та time. Робота з випадковими величинами. Модуль math.

Робота з датою

Модуль **datetime** надає класи для маніпуляцій з датами і часом.

Ключові аспекти: методи роботи з датами й часом

Імпорт модуля:

import datetime

Поточна дата і час (метод datetime.**datetime.now()**):

import datetime

now = datetime.datetime.now()

print(now) *#у форматі рік-місяць-день години:хвилини:секунди.мікросекунди*

Або

from datetime import datetime

now = datetime.now()

print(now.year) *# рік*

print(now.month) *#місяць*

print(now.day) *# день*

print(now.hour) *# година від 0 до 23*

print(now.minute) *#хвилини від 0 до 59*

print(now.second) *# секунди від 0 до 59*

print(now.microsecond) *# мікросекунди від 0 до 999999*

print(now.tzinfo) *#часова зона. Якщо не була вказана, то буде None.*

print(now.date()) *# дата (без часу)*

print(now.time()) *# час (без дати)*

print(now.weekday()) *# номера дня тижня для дати. (понеділок – 0, а неділя – 6)*

Є зворотний метод **datetime.combine(date\_object, time\_object)** який використовується для створення нового об'єкта **datetime** шляхом комбінування об'єктів **datetime.date()** та **datetime.time()**. Це дозволяє створювати точний момент часу, вказуючи дату та час окремо, а потім об'єднуючи їх.

import datetime

*# Створення об'єктів date і time*

date\_part = datetime.date(2023, 12, 14)

time\_part = datetime.time(12, 30, 15)

*# Комбінування дати і часу в один об'єкт datetime*

combined\_datetime = datetime.datetime.**combine(**date\_part, time\_part**)**

Для створення об'єкта **datetime** з певною датою:

specific\_date = datetime.datetime(year=2020, month=1, day=7)

Створення об'єкта datetime з датою та часом

specific\_datetime = datetime.datetime(year=2020, month=1, day=7, hour=14, minute=30, second=15)

Методи порівняння об'єктів datetime:

Для порівняння двох об'єктів **datetime** (**==**, **!=**, **<**, **>**, **<=**, **>=**).

*# Порівняння дат*

print(datetime1 == datetime2) # False, тому що дати не однакові

print(datetime1 != datetime2) # True, тому що дати різні

print(datetime1 < datetime2) # True, тому що datetime1 передує datetime2

print(datetime1 > datetime2) # False, тому що datetime1 не наступає за datetime2

Робота з часовими проміжками timedelta

У модулі **datetime** є клас **timedelta**, який використовується для представлення різниці між двома моментами в часі.

from datetime import timedelta

delta = timedelta(days=50, seconds=27, microseconds=10, milliseconds=29000, minutes=5, hours=8, weeks=2)

print(delta)

Якщо відняти від одного datetime об'єкту інший, то отримаємо timedelta об'єкт. Він відповідає за відрізок часу між двома датами.

Метод .total\_seconds() виконали конвертацію timedelta в секунди.

Об'єкти timedelta можна створювати, щоб отримати час / дату, віддалену від початкової. Або від якоїсь конкретної дати.

future\_date = now + timedelta(days=10) # Додаємо 10 днів до поточної дати

Визначення кількості днів між двома датами, метод **.toordinal()** (перетворює об'єкт **datetime** в ціле число).

Робота з timestamp

**timestamp** представляється як кількість секунд, з UTC + 0. (часовий пояс)

У Python ви можете перетворити об'єкт **datetime** в **timestamp** і навпаки. Конвертація datetime в **timestamp**

timestamp = **datetime.timestamp(**now**)**

Конвертація **timestamp** в об'єкт datetime

dt\_object = **datetime.fromtimestamp(**timestamp**)**

Парсинг дати із рядка

**strftime** використовується для форматування об'єктів дати та часу **datetime** у рядки за допомогою специфічних форматних кодів.

Синтаксис методу **strftime** виглядає наступним чином:

datetime\_object.strftime("format")

Ось деякі з найбільш використовуваних кодів:

* **%Y** - рік з чотирма цифрами (наприклад, **2023**).
* **%y** - рік з двома цифрами (наприклад, **23**).
* **%m** - місяць як номер (наприклад, **03** для березня).
* **%d** - день місяця як номер (наприклад, **14**).
* **%H** - година (24-годинний формат) (наприклад, **15**).
* **%I** - година (12-годинний формат) (наприклад, **03**).
* **%M** - хвилини (наприклад, **05**).
* **%S** - секунди (наприклад, **09**).
* **%A** - повна назва дня тижня (наприклад, **Tuesday**).
* **%a** - скорочена назва дня тижня (наприклад, **Tue**).
* **%B** - повна назва місяця (наприклад, **March**).
* **%b** або **%h** - скорочена назва місяця (наприклад, **Mar**).
* **%p** - AM або PM для 12-годинного формату.

Метод **strptime** в Python використовується для перетворення рядків у об'єкти **datetime**. (протилежність до **strftime**).

Синтаксис методу **strptime** виглядає наступним чином:

datetime\_object = datetime.**strptime**(string, format)

* Коди форматування для **strptime** такі ж, як і для **strftime**. Він дозволяє перетворити рядки на об'єкти **datetime**.

Робота з ISO форматом дати

Формат дати в ISO 8601 виглядає як "YYYY-MM-DD", де:

* **YYYY** - це рік (наприклад, 2023),
* **MM** - місяць (наприклад, 01 для січня),
* **DD** - день (наприклад, 31).

Формат часу в ISO 8601 виглядає як "HH:MM:SS", де:

* **HH** - години (від 00 до 23),
* **MM** - хвилини (від 00 до 59),
* **SS** - секунди (від 00 до 59, іноді з мікросекундами).

Повне представлення дати та часу в ISO 8601 поєднує ці два формати з "T" між ними, наприклад "YYYY-MM-DDTHH:MM:SS".

"Z" на кінці означає UTC (координований всесвітній час), а відхилення від UTC "+HH:MM" або "-HH:MM".

Ключові аспекти: методи для роботи з ISO форматом дати

* Об'єкт **datetime** можна перетворити в рядок формату ISO 8601 за допомогою методу **isoformat()**:

✂️ Цей код можна запустити!

iso\_format = now.**isoformat()**

* Для зворотного перетворення рядка у форматі ISO 8601 на об'єкт **datetime**, можна використати метод **fromisoformat()**:

date\_from\_iso = datetime**.fromisoformat(**iso\_date\_string**)**

* Метод **isoweekday()** у об'єкті **datetime** використовується для отримання дня тижня відповідно до ISO 8601. Понеділок - 1, неділя - 7.

day\_of\_week = now.isoweekday()

* Виведення методу **.isocalendar()** - це кортеж **(ISO\_рік, ISO\_тиждень, ISO\_день\_тижня)**, де:
* **ISO\_рік** - це рік у форматі ISO.
* **ISO\_тиждень** - номер тижня в році за ISO 8601 (від 1 до 53).
* **ISO\_день\_тижня** - день тижня за ISO 8601, де 1 означає понеділок, а 7 - неділю.

Робота з часовими зонами

Щоб вивести дату у форматі UTC треба додати інформацію про часову зону до об'єкта **datetime**:

utc\_now = datetime.now(**timezone.utc**) *# Виведе поточний час в UTC*

Щоб перетворити час з UTC в іншу часову зону, вам знадобиться визначити об'єкт **timezone** з відповідним зсувом. Наприклад, для перетворення UTC часу в час, що відповідає Східному часовому поясу США (UTC-5 годин), можна зробити наступне:

utc\_time = datetime.now(timezone.utc)

eastern\_time = utc\_time.**astimezone**(timezone(timedelta(hours=-5)))

*# Виведення: Поточний\_час-05:00*

Щоб перетворити локальний час у час UTC, спочатку потрібно призначити

локальному часу відповідну часову зону, а потім використати метод **astimezone()** для конвертації його в UTC:

*# Припустимо, місцевий час належить до часової зони UTC+2*

local\_timezone = timezone(timedelta(hours=2))

local\_time = datetime(….., tzinfo=local\_timezone)

*# Конвертація локального часу в UTC*

utc\_time = local\_time.**astimezone**(timezone.utc)

Стандарт ISO 8601 також підтримує часові зони. У Python це можна зробити, додавши інформацію про часову зону до об'єкта **datetime**:

from datetime import datetime, timezone, timedelta

*# Час у конкретній часовій зоні*

timezone\_offset = timezone(timedelta(hours=2)) # Наприклад, UTC+2

timezone\_datetime = datetime(….., tzinfo=timezone\_offset)

*# Конвертація у формат ISO 8601*

iso\_format\_with\_timezone = timezone\_datetime**.isoformat()**

Робота з часом

Розглянемо основні методи модуля time

Метод **time.time()** повертає поточний час у секундах з 1 січня 1970 року.

Метод **time.sleep(seconds)** зупиняє виконання програми на вказану кількість секунд.

Метод **time.ctime([seconds])** перетворює часову мітку (кількість секунд) у зрозуміле для людини текстове представлення. Якщо аргумент не вказаний, використовується поточний час.

current\_time = time.time()

readable\_time = time.ctime(current\_time)

Метод **time.localtime([seconds])** перетворює часову мітку в структуру **struct\_time** у місцевій часовій зоні.

current\_time = time.time()

local\_time = time.localtime(current\_time) # Виведення: time.struct\_time(tm\_year, tm\_mon, tm\_mday, tm\_hour, ….)

Метод **time.gmtime([seconds]) с**хожий на **localtime**, але повертає **struct\_time** у UTC.

Метод **time.perf\_counter(),**надає доступ до лічильника з високою точністю, та є ідеальним для вимірювання коротких інтервалів часу(повертає значення в секундах).

execution\_time = end\_time - start\_time

Щоб зробити великі числа більши читабельними, використовують **\_**. Наприклад, число **1\_000\_000** еквівалентне **1000000**.

Робота з випадковими величинами

* **random.randint(a, b):** Отримання випадкового цілого числа з рівномірного розподілу в інтервалі між a та b включно.
* **random.random():** Отримання випадкового числа в інтервалі між 0.0 (включно) та 1.0 (не включно).
* **random.randrange(start, stop[, step]):** Отримання випадкового числа з заданого діапазону, з можливістю вказати крок між значеннями. stop не включається
* **random.shuffle(x):** Перемішування порядку елементів у списку x.
* **random.choice(seq):** Вибір випадкового елемента з послідовності seq (списку або кортежу).
* **random.choices(population**(кортеж)**, weights=None**(кортеж вагів, кількість елементів, як в population)**, cum\_weights=None, k=1**(кількість елем.)**):** Генерація випадкової вибірки з можливістю зазначити ймовірності для кожного елемента та повторення у вибірці.
* **random.sample(population, k):** Отримання унікальних випадкових елементів зі списку population довжиною k.
* **random.uniform(a, b):** Отримання випадкового дійсного числа N такого, що a <= N <= b.

Модуль math

import math

Константи:

* math.pi - константа π*π* (приблизно 3.14159...);
* math.e - константа e*e*, основа натуральних логарифмів (приблизно 2.71828...);
* math.tau - константа τ*τ*, дорівнює 2π2*π* (приблизно 6.28318...);
* math.inf - позначення нескінченності;
* math.nan - позначення 'Not a Number' (не число);

Функції округлення чисел:

* **math.ceil(x) -** виконує округлення дійсного числа x до найближчого більшого цілого числа;
* **math.floor(x) -** виконує округлення дійсного числа x до найближчого меншого цілого числа;
* **math.trunc(x)** - виконує обрізання дробової частини дійсного числа x, та повертає цілу частину числа;

Тригонометричні функції

* **math.sin(x) -** синус x, де x в радіанах;
* **math.cos(x) -** косинус x;
* **math.tan(x) - т**ангенс x;
* **math.asin(x) -** арксинус x;
* **math.acos(x) -** арккосинус x;
* **math.atan(x)** - арктангенс x;

Експоненційні та логарифмічні функції

* **math.exp(x)** - число e*e* в ступені x;
* **math.log(x[, base])** - Логарифм x за основою base. Якщо base не вказано, обчислюється натуральний логарифм;

Ступінь та корінь

* **math.pow(x, y)** - x у ступені y;
* **math.sqrt(x)** - квадратний корінь з x;

Деякі інші функції

* **math.fabs(x)** - модуль (абсолютне значення) x;
* **math.factorial(x)** - факторіал числа x;
* **math.gcd(x, y) -** найбільший спільний дільник для x та y;

Якщо вам потрібна комплексна математика, то можна скористатися пакетом cmath. Він надає той самий API, але вміє працювати з комплексними числами.

Правильне порівняння дійсних чисел

Дійсні числа в комп'ютерних програмах часто можуть викликати неточності через їхню бінарну природу.

print(0.1 + 0.2) *# Виведення буде: 0.30000000000000004*

Функція **math.isclose** використовується для порівняння двох чисел з певною допустимою похибкою.

r = math.isclose(0.1 + 0.2, 0.3) *# Це поверне True*

r = math.isclose(0.1, 0.10000000009) *# Це поверне True*

Регулярні вирази та розширена робота з рядками

****Просунута робота з рядками****

Коли в тексті більше одного рядка можна скористатися потрійним повторенням лапок: """ """

Символ \ в кінці рядка коду вказує інтерпретатору ігнорувати закінчення рядка і продовжити відразу з наступного.

У Python, коли ви поміщаєте два рядкових літерали поруч, вони автоматично конкатенуються (об'єднуються в один рядок). Це неявна конкатенація рядків

("spam " "eggs") == "spam eggs" # True

****Спеціальні символи****

Ці символи починаються з (backslash, **\**), після якої йде один або кілька символів, що визначають спеціальну послідовність.

Основні керувальні символи які необхідно знати:

* **\n** відповідає за перенос рядка (line break).
* **\f** відповідає за перенос сторінки.
* **\t** горизонтальна табуляція (tab).
* **\v** вертикальна табуляція (tab).
* коли ми зустрічаємо символ **\r**, то повертаємося на початок рядка і продовжуємо виведення.
* Керувальний символ **\b** — забій (backspace). Видаляє символ попереду
* Виведення зворотної скісної риски. "Hello\\World"
* Щоб екранувати одинарні та подвійні лапки.

print('It\'s a beautiful day')

print("He said, \"Hello\"")

****Методи рядків****

****Пошук у рядку****

Для пошуку деякого символу або підрядка у рядку можна скористатися методом find:

print(рядок.find("er", start, end)) # 5

Цей метод повертає індекс початку першого збігу в рядку, починаючи з start до end. Якщо start та end не вказані, то пошук відбувається з початку і до кінця рядку. Метод повертає -1, якщо послідовність не знайдена.

Метод схожий на find — це index. Але якщо index не знайде підрядок, то викличе виняток ValueError.

Якщо вам потрібно здійснити пошук підрядка у рядку справа, а не зліва як у find, то для цього існує метод .rfind("…"). І "правий" аналог index — rindex

****Поділ та об'єднання рядків****

*Усі рядки незмінні, і якщо ми хочемо модифікувати рядок, є тільки один спосіб — створити новий рядок на основі вихідного.*

Метод **split()** використовується для розбиття рядка на список підрядків на основі вказаного роздільника. (за замовчуванням використовується пробіл)

Синтаксис методу split()

new = str.split(separator=None, maxsplit=-1)

**separator** – роздільник, **maxsplit -** максимальна кількість розділень. (-1 без обмежень).

Метод **join()** використовується для об'єднання послідовності рядків (наприклад, списку або кортежу) в один рядок з використанням вказаного роздільника.

Синтаксис методу join()

new = string.join(iterable)

**string -** рядок-роздільник (н.п. ', '), **iterable –** послідовність.

Якщо потрібно видалити зайві пробіли на початку і в кінці рядка, є спеціальний метод strip, lstrip, видаляє пробіли на початку рядка, rstrip видаляє пробіли в кінці рядка

clean = ' spacious '.strip()

Метод **replace()** у Python використовується для заміни підрядка на інший підрядок у рядку. Цей метод повертає новий рядок, де кожне входження вказаного підрядка замінено на інший підрядок.

Метод replace() має наступний синтаксис

new = str.replace(old, new, count=-1)

**old -** підрядок, який потрібно замінити, **new** - підрядок, на який потрібно замінити (якщо "", то видаляться входження з рядка), **count** - лічильник максимальної кількості замін. Якщо не вказано або вказано -**1**, замінюються всі входження.

Для видалення фіксованої послідовності на початку рядка є метод removeprefix:

print('TestHook'.removeprefix('Test')) # Hook

print('TestHook'.removeprefix('Hook')) # TestHook

Є парний метод для видалення послідовності в кінці рядка, removesuffix:

print('TestHook'.removesuffix('Test')) # TestHook

print('TestHook'.removesuffix('Hook')) # Test

Символ **\_** використовується для ігнорування частини, яка нам не потрібна, після розділення рядка. \_, змінна = текст.split('знак\_розділення') (якщо знак ділить на 2 частини, і нам потрібна тільки 2)

Метод **.isdigit()** використовується для перевірки, чи складається рядок повністю з цифр.

****Translate****

Метод **translate()** використовується для перетворення рядків шляхом заміни певних символів іншими символами. Перш ніж використовувати **translate()**, нам потрібно створити таблицю перекладу. Це можна зробити за допомогою методу **str.maketrans()**, який приймає два аргументи(або словник):

1. Рядок символів, які потрібно замінити. В Unicode коді (без методу **str.maketrans() використовувати ->** **ord("…")**)
2. Рядок символів, на які потрібно замінити.

Довжина обох рядків має бути однаковою, оскільки вони визначають взаємну відповідність символів.

Метод **translate()** також може використовуватися для видалення певних символів із рядка. Для цього передайте в **maketrans()** третій аргумент - рядок символів, які потрібно видалити.

trantab = str.maketrans('', '', символи\_які\_треба\_видалити)

print(str.translate(trantab))

[Форматування рядків](https://docs.python.org/3/library/string.html#format-specification-mini-language)

Вивести числа в десятковому, шістнадцятковому, вісімковому і двійковому представленні можна наступним чином:

s = f"int: {число:d}; hex: {число:#x}; oct: { число:#o}; bin: {число:#b}"

total = f"Total: {число:.2f}" ***# :.2f*** *для відображення дійсного числа з двома цифрами після крапки.*

У виразі **:.2f**:

* **:** вводить специфікацію формату.
* **.2** означає, що після десяткової крапки має бути виведено дві цифри.
* **f** вказує на формат дійсного числа.

Відцентрувати значення у стовпцях по 10 символів (на кожне число) шириною:

f'{num:^**10**} {num\*\*2:^**10**} {num\*\*3:^**10**}'

0 0 0

Можливі варіантио вирівнювання:

* **<** Вирівнювання вмісту по лівому краю.
* **>** Вирівнювання вмісту по правому краю.
* **^** Вирівнювання вмісту по центру.
* **=** Використовується для вирівнювання чисел, при цьому знак (якщо він є) відображається зліва, а число - по правому краю поля.

Форматування відсотків у f-рядках виглядає так:

f"{value:<ширина>.<точність>%}"

****Регулярні вирази****

[Регулярні вирази](https://uk.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B3%D1%83%D0%BB%D1%8F%D1%80%D0%BD%D0%B8%D0%B9_%D0%B2%D0%B8%D1%80%D0%B0%D0%B7) - дозволяють шукати, замінювати або витягувати певні шаблони в тексті за допомогою спеціального синтаксису.

Регулярні вирази дозволяють шукати певні шаблони в рядках, виконувати заміни, розбивати рядки на частини і багато іншого. Для роботи з регулярними виразами в Python використовується модуль **re**.

**pattern – починається з -> r"…"**

Приклади **блоку** можуть бути:

* \w — будь-яка цифра або буква [a-zA-Z0-9\_]

(\W — все, крім букви або цифри [^a-za-z0-9\_])

* \d — будь-яка цифра [0-9]

(\D — усе, крім цифри [^0-9])

* \s — будь-який пробільний символ [\t\n\r\f\v]

(\S — усе, крім пробільних символів [^\t\n\r\f\v])

* \b — межа слова
* [...] — один із символів у дужках

([^ ] — будь-який символ, крім тих, що в дужках)

* ^ і $ — початок і кінець рядка відповідно
* ( ) — групує вираз і повертає знайдений текст
* \t, \n, \r — символ табуляції, нового рядка та повернення каретки

**Модифікатори** можуть вказувати на кількість повторень блоку у виразі, наприклад:

* . — один будь-який символ, крім рядка \n
* ? — 0 або 1 входження шаблону зліва
* + — 1 і більше входжень шаблону зліва
* \* — 0 і більше входжень шаблону зліва
* \ — екранування спеціальних символів (приклад: \. — означає крапку або \+ — знак "плюс")
* {n} суворо n разів (n ціле число)
* {n,m} — від n до m входжень (приклад: {,m} — від 0 до m)
* a|b — відповідає a або b. Сам символ | означає "або" між двома шаблонами
* ( ) — групує вираз і повертає знайдений текст

import re

Загальним для усіх функцій модуля **re** є те, що першим аргументом йде регулярний вираз у вигляді рядка.

Основні функції модуля re:

****Метод search****

**re.search()** використовується для пошуку першого входження шаблону в рядку.

re.search(pattern, string)

Результат виконання **re.search()** це спеціальний об'єкт **Match**, якщо знаходить відповідність. Якщо відповідність не знайдена, повертає **None**.

Об'єкт **Match** має методи, для отримання інформації про пошук та результат:

* Match.span() повертає кортеж, що містить початкову та кінцеву позиції збігу.
* Match.string повертає рядок, переданий у функцію,
* Match.group() повертає частину рядка, в якому був збіг

В функцію **search** ми передаємо параметр **re.IGNORECASE**, який робить пошук нечутливим до регістру. А отже слово може бути як з великих так і малих літер.

Треба розділити "username@domain.com" на "username" (ім'я користувача) та "domain.com" (домен).

✂️ Цей код можна запустити!

pattern = r"(\w+)@(\w+\.\w+)"

match = re.search(pattern, email)

if match:

user\_name = match.group(1)

domain\_name = match.group(2)

Методи **match.group(1)** та **match.group(2)** використовуються для вилучення тексту, що відповідає кожній групі в шаблоні.

****Метод findall****

Метод **re.findall()** використовується для знаходження всіх входжень шаблону, заданого регулярним виразом, у заданому рядку.

matches = re.findall(pattern, string)

****Метод sub****

Метод **re.sub()** в модулі **re** Python використовується для заміни входжень регулярного виразу **pattern** в рядку **string** на рядок **repl**.

Метод повертає рядок, у якому всі входження **pattern** замінені на **repl**.

pattern = r"(\d{3})-(\d{3})-(\d{4})" *# 000-000-0000 групи\1\2\3*

replacement = r"(\1) \2-\3" *# (000) 000-0000 використ. Групи з pattern*

****Метод split****

Функція **re.split()** в модулі **re** Python використовується для розбивання рядка за заданим регулярним виразом.

Синтаксис:

list\_of\_elements = re.split(pattern, string)

Python Core 2.0 ֍ 4)Робота з файлами та модульна система

Робота з файлами та обробка винятків

Python надає функціонал роботи з будь-якими файлами.

Відкрити файл:

fh = open('test\_file.txt')

fh — це файловий об'єкт, через який ми можемо працювати з файлом.

Після того, як робота з файлом завершена потрібно викликати метод **close**:

fh.close()

**Закривати файл обов'язково.** Якщо файлу з ім'ям test\_file.txt в системі немає, то ви отримаєте виняток.

Після короткого огляду, розглянемо синтаксис функції **open()** більш детально. Функція **open()** повертає файловий об'єкт, який далі може бути використаний для читання з файлу або запису в нього.

open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None)

Ось основні режими **mode**, які визначають як буде відкрито файл:

* **'r'** - читання (за замовчуванням). Файл має існувати.
* **'w'** - запис. Створює новий файл або перезаписує той, що вже існує.
* **'a'** - додавання. Дописує в кінець файлу, не перезаписуючи його.
* **'b'** - бінарний режим (може бути використаний разом з іншими, наприклад **'rb'** або **'wb'**).
* **'+'** - оновлення (читання та запис).

fh.write('hello!') *# запису даних у файл*

fh.seek(0) *# повернути курсор на початок файлу*

print(fh.tell()) *# дізнатися положення курсора*

symbols = fh.read(2) *# дозволяє прочитати деяку кількість символів з файлу # 'he'*

Щоб прочитати увесь вміст файлу за раз, можна викликати метод read без аргументів.

Читати файл по одному рядку за раз можна методом readline з використанням циклу

Метод readlines читає увесь файл повністю, але повертає список рядків, де елемент списку — це один рядок з файлу.

lines = fh.readlines()

Всі методи, які читають файли порядково, залишають (не видаляють) символ перенесення рядка \n. Його, за необхідності, треба видаляти методом **strip**()

Обробка винятків. Менеджер контексту

Щоб уникнути помилок, можна взяти блок коду, в якому відбувається робота з файлом, у блок try ... except:

fh = open('text.txt', 'w')

try:

*# Виконання операцій з файлом*

finally:

*# Закриття в блоці finally гарантує, що файл закриється навіть у разі помилки*

fh.close()

Для покращення читабельності коду при збереженні функціоналу можна скористатися менеджером контексту **with**.

with open('text.txt', 'w') as fh:

*# Виконання операцій з файлом*

*# Файл автоматично закриється після виходу з блоку with*

Робота з нетекстовими файлами у Python

with open('raw\_data.bin', 'wb') as fh:

fh.write(b'Hello world!')

В режимі  **wb** у файл можна писати тільки байт-рядки або байт-масиви.

Для байт-рядків застосовуються ті самі обмеження і правила, що і для звичайних рядків. методи **upper()**, **startswith()**, **index()**, **find()** і так далі.

**ASCII** (Американський стандартний код для обміну інформацією) - це символьна кодова таблиця, яка використовується для представлення тексту в комп'ютерах, комунікаційному обладнанні та інших пристроях, що працюють з текстом. Кожен символ у таблиці ASCII відповідає певному числу.

Створимо наступний байт-рядок:

byte\_string = b'Hello world!'

Другий спосіб створення байт рядків — це перетворення у байт-рядок.

Для перетворення рядка у байт-рядок можна скористатися методом **.encode()**

byte\_str = 'some text'.encode()

print(byte\_str)

Перетворення чисел у байт-рядки

У Python ви можете перетворювати числа у байт-рядки за допомогою вбудованої функції **bytes**.

numbers = [0, 128, 255]

byte\_numbers = bytes(numbers)

Щоб перевірити правильність представлення, можна скористатися вбудованою функцією hex, яка перетворить ціле число в рядок — представлення числа в шістнадцятковій формі:

for num in [127, 255, 156]:

print(hex(num))

Кодування рядків (ASCII, UTF-8, CP1251)

Щоб дізнатися, якому елементу в UTF-8 відповідає символ, є функція **ord("…")**.

Зворотна операція, коли потрібно дізнатися, який символ закодований числом, **chr(…)**

Python може працювати з дуже великою кількістю різних кодувань.

utf8 = текст.encode()

utf16 = текст.encode("utf-16")

cp1251 = текст.encode("cp1251")

s\_from\_utf16 = utf16.**decode**("utf-16")

Щоб уникнути проблем з кодуванням можна використати **encoding** у функції **open()**.

with open('example.txt', 'r', encoding='utf-8') as file:

content = file.read()

print(content)

Навіть якщо операційна система використовує інше кодування за замовчуванням, файл буде коректно відкритий із використанням UTF-8, що гарантує правильне відображення тексту.

Масив байтів

Робота з рядками обмежена тим, що рядки і байт-рядки незмінні. Якщо потрібно замінити навіть один символ, потрібно, по суті, створити копію початкового рядка з єдиним відмінним символом. Щоб зменшити накладні витрати при роботі з "сирими" даними, в Python є такий контейнер як **bytearray**.

byte\_array = bytearray(b'Kill Bill')

byte\_array[0] = ord('B')

byte\_array[5] = ord('K')

print(byte\_array) # Виведення: bytearray(b'Bill Kill')

В іншому ж bytearray може використовуватися як заміна байт-рядків і у нього є ті самі методи з тією самою поведінкою.

Окрім зміни наявних елементів, **bytearray** дозволяє додавати та видаляти елементи, що робить його набагато більш гнучким у порівнянні з незмінними байт-рядками.

byte\_array = bytearray(b"Hello")

byte\_array.**append**(ord("!"))

Хоча **bytearray** сприймається як послідовність чисел, його можна легко перетворити в рядок за допомогою методу **decode()**, вказавши потрібне кодування.

byte\_array = bytearray(b"Hello World")

string = byte\_array.decode("utf-8")

print(string) # Виведе: 'Hello World'

Порівняння рядків

Основні кроки процесу порівняння наступні.

* Перетворення рядків у єдиний регістр (нижній - **.lower()**, верхній - **.upper()**)
* Далі порівняння. (==)

Щоб розв'язати цю проблему при роботі з не ASCII символами для порівняння рядків, їх необхідно нормалізувати за допомогою методу **.casefold()**, який повертає рядок, де всі символи у нижньому регістрі і без неоднозначностей, коли будь-який символ матиме тільки одну можливу форму запису. **casefold()** надає додаткову точність для специфічних мовних випадків.

print(text.casefold())

Робота з архівами

Архіви по своїй суті — це ті самі файли, але інформація в них розташована з використанням алгоритмів стискання, які дозволяють записати інформацію в меншому об'ємі.

Пакет shutil вміє працювати з архівами. (копіювання, переміщення, перейменування та видалення файлів і директорій) зручніший, ніж модуль **os**.

import shutil

Пакет shutil підтримує архіви zip, tar, gz. Для цього він використовує пакети zipfile та tarfile.

Функція **shutil.make\_archive()**- створення архівів (наприклад, ZIP або TAR файлів) з заданої директорії.

shutil.make\_archive(шлях, format, root\_dir=None, base\_dir=None)

* **root\_dir -** директорія, з якої буде створено архів. Якщо не вказано, використовується поточна директорія. Запакує всі файли та піддиректорії, що знаходяться у цій директорії.
* **base\_dir -** директорія всередині архіву, з якої почнеться архівація.

Розпаковування архівів - функція **shutil.unpack\_archive()**

shutil.unpack\_archive(шлях, extract\_dir=None, format=None)

* **extract\_dir -** директорія, куди буде розпаковано вміст архіву. Якщо не вказано, використовується поточна директорія.

Окрім роботи з архівами модуль shutil може ще виконувати наступні високорівневі операції для обробки файлової системи:

* **shutil.copy(src, dst)** копіює файл з **src** в **dst**. Якщо **dst** є директорією, файл буде скопійований зі своїм поточним іменем у цю директорію.
* **shutil.copytree(src, dst)** рекурсивно копіює всю директорію **src** в директорію **dst**.
* **shutil.move(src, dst)** переміщує файл або директорію **src** в **dst**.
* **shutil.rmtree(path)** рекурсивно видаляє директорію **path**.
* **shutil.disk\_usage(path)** повертає статистику використання диска, що містить загальний об'єм, використаний об'єм і вільний об'єм для даного шляху.

Основи модуля pathlib

Модуль **pathlib** в Python є сучасним інструментом для роботи з файловою системою. (прийшов на заміну застарілому модулю **os**)

Два основних класи у цьому модулі - це **Path** та **PurePath**.

**PurePath** - маніпуляція шляхами без доступу до файлової системи. **PurePath** (розділення шляху на частини, перевірка суфіксів, імен файлів, шляхів тощо).

from pathlib import PurePath

p = PurePath("/usr/bin/simple.jpg")

print("Name:", p.name) # Name: simple.jpg

print("Suffix:", p.suffix) # Suffix: .jpg

print("Parent:", p.parent) # Parent: \usr\bin

**Path(\_\_file\_\_)** – шлях до поточного файла

Клас **Path** наслідує всі методи з **PurePath** + (читання, запис файлів, перевірка існування файлів, перелічувати файли у директорії тощо).

Path варто сприймати як вказівку на файл або директорію.

from pathlib import Path

p = Path("example.txt")

p.write\_text("Hello, world!")

print(p.read\_text()) # Hello, world!

print("Exists:", p.exists()) # Exists: True перевірки існування файлу

Створення Шляхів

Клас **Path** автоматично адаптується до особливостей шляхів у різних операційних системах. Наприклад, у Windows використовуються зворотні слеші (**\**), тоді як в Unix-подібних системах (Linux, macOS) - прямі слеші (**/**).

base\_path = Path("/usr/bin") *# Початковий шлях*

*# Додавання додаткових частин до шляху*

full\_path = base\_path / "subdir" / "script.py"

print(full\_path) *# Виведе: /usr/bin/subdir/script.py*

Відносні та абсолютні шляхи

**Абсолютний шлях** - це повний шлях до файлу або директорії від кореня файлової системи. C:\Users\user\documents\example.txt

**Відносний шлях** - це шлях до файлу або директорії відносно поточного робочого каталогу.

*# Перетворення відносного шляху в абсолютний*

relative\_path = Path("documents/example.txt")

absolute\_path = relative\_path.**absolute()**

Метод **.relative\_to()** використовується для отримання відносного шляху щодо заданої директорії.

Маніпуляція з компонентами шляху

Клас **Path** модуля **pathlib** надає зручні методи для маніпулювання компонентами шляху (**.with\_name("…")** - дозволяють легко змінювати ім'я файлу, **.with\_suffix("…")** - замінює або додає розширення файлу н.п. ".txt")

Методи **with\_name** і **with\_suffix** не змінюють фізичне ім'я файлу на диску.

Щоб фізично змінити ім'я файлу на диску, потрібно використовувати методи для роботи з файловою системою, наприклад, **rename.**

# Створює новий об'єкт Path з іншим ім'ям файлу

new\_path = original\_path.with\_name("…")

original\_path.**rename(**new\_path**)**

Читання та запис файлів

Модуль **pathlib** надає кілька методів для читання з файлів та запису в них, але вони не замінюють модуль **open**, а є доповненням.

Методи **read\_text()** та **write\_text()** використовуються для та запису текстових файлів.

Path.read\_text(encoding=None, errors=None) *# читання*

Path.write\_text(data, encoding=None, errors=None) *# запис*

Параметр **errors**, в обох методах, визначає, як мають бути оброблені ці помилки.

Методи **read\_bytes()** та **write\_bytes()** використовуються для читання та запису бінарних файлів.

Роботу з директоріями (Модуль pathlib)

Метод **iterdir()** використовується для отримання переліку всіх файлів та піддиректорій у вказаній директорії.

*# Виведення переліку всіх файлів та піддиректорій*

for path in directory.iterdir():

print(path)

Для створення нової директорії використовується метод **mkdir()**.

Path.mkdir(mode=0o777, parents=False, exist\_ok=False)

* **mode -** використовуються для Linux і не актуальні для Windows.
* **parents -** якщо має значення **True**, створить всі батьківські директорії, які відсутні.
* **exist\_ok -** якщо має значення **True**, помилка не буде викинута, якщо директорія вже існує.

Метод **rmdir()** видаляє директорію, але директорія повинна бути порожньою.

* метод **exists()** перевіряє, чи існує файл або директорія.
* метод **is\_dir()** перевіряє, чи є об'єкт директорією.
* метод **is\_file()** перевіряє, чи є об'єкт файлом.

if path.метод():

print(f"{path} існує")

Переміщення та копіювання файлів

Модуль **pathlib** інтегрується з модулем import **shutil** для копіювання та переміщення файлів. Для копіювання - функція **shutil.copy()** або **shutil.copy2()**.

*# Копіювання файла Path(source) в файл Path(destination)*

shutil.copy(source, destination)

Функція **shutil.copy()** копіює вміст файлу, але не копіює метадані, тоді як **shutil.copy2()** копіює і вміст, і метадані.

Для переміщення файлів використовується функція **shutil.move(**source, destination**)**.

Метод **stat()** повертає інформацію про файл, включаючи його розмір.

*# Отримання розміру файла в байтах*

size = file\_path.stat()**.st\_size**

Метод **stat()** також надає час створення, атрибут **.st\_ctime** , і час останньої модифікації файлу, атрибут **.st\_mtime**.

Для видалення файлу використовується метод **unlink()**. Він видаляє файл, на який вказує об'єкт **Path**.

Path.unlink(missing\_ok=False)

Робота з модулями та створення віртуального оточення

Імпорт пакетів та модулів

Модуль як файл(містить різні функції, класи, змінні), а пакет набір файлів - директорію з файлами.

Імпортувати пакет, можна за допомогою ключового слова **import**, після якого ви можете вказати один або декілька пакетів, які ви хочете імпортувати.

Для того щоб викликати функцію з пакету, потрібно вказати ім'я пакету і через крапку ім'я функції або константи в цьому пакеті. Є й інший спосіб: можна імпортувати те, що нам необхідно за допомогою виразу **from ... import ...**:

☝ Можна імпортувати весь вміст модуля за допомогою інструкції **from <module\_name> import \***. (не рекомендується через можливі конфлікти імен)

Імпортувати код можна не лише із стандартних або встановлених пакетів (модулів), але й з власноручно написаних модулів Python.

Що таке модуль у Python?

Модулем Python є будь-який текстовий файл з розширенням **py**, який містить код мовою Python. (нп функція, яку можна викликати в іншому файлі)

Для використання створеного модуля в іншому файлі Python, ми імпортуємо його

Ми також можемо змінити назву функції say\_hello всередині main.py, коли імпортуємо її з модуля mymodule.py

*# main.py*

from mymodule import say\_hello as greeting *# надаємо функції псевдонім greeting*

Функція dir()

**dir(module\_name)**: Повертає список атрибутів, визначених у модулі **module\_name**. (Без аргументів, повертає список атрибутів з поточної області визначень.)

Службова змінна \_\_name\_\_

Важливо розуміти, що під час імпорту модуля Python виконує увесь код, що міститься в модулі.

Далеко не завжди така поведінка бажана.

В таких випадках нам може допомогти службова змінна Python: \_\_name\_\_.

Тоді під час імпорту функції із mymodule.py код у блоці if \_\_name\_\_ == '\_\_main\_\_': не буде виконаний, а якщо ж в консолі виконати **python mymodule.py**, то буде.

Для зручності прийнято весь код, який потрібно виконати з модуля, поміщати у функцію main:

def say\_hello(name):

print(f'Hello {name}')

def main():

print("You imported hello.py")

say\_hello('user')

if \_\_name\_\_ == '\_\_main\_\_':

main()

Модуль sys. Обробка аргументів командного рядка

Модуль **sys** часто використовується для отримання інформації про середовище, в якому працює програма, і для взаємодії з системою на нижчому рівні.

Ключові можливості модуля **sys**:

1. **sys.argv -** список аргументів командного рядка, переданих скрипту Python. Елемент **argv[0]** є ім'ям скрипту, а інші елементи – це додаткові аргументи командного рядка.
2. **sys.exit() -** функція виходу з Python. Ви можете передати числовий аргумент, що стане статусом виходу програми. Прийнято, що аргумент 0 означає успішне завершення, а ненульові значення вказують на помилку.
3. **sys.path -** список рядків, визначає шлях пошуку інтерпретатора для модулів. Ви можете модифікувати цей список, щоб додати власні шляхи для пошуку модулів.
4. **sys.version -** рядок, що містить інформацію про версію Python, яка використовується.
5. **sys.platform -** рядок, що вказує на ім'я платформи, на якій виконується Python (наприклад, 'linux' для Linux, 'win32' для Windows).
6. **sys.modules[“…”] -** словник, який містить завантажені модулі. Ключі – це назви модулів, а значення – це об'єкти модулів. Виведенням буде шлях, де знаходиться модуль **“…”**. Виклик sys.modules.keys() поверне повний список імен завантажених модулів.
7. **sys.builtin\_module\_names** - список вбудованих модулів, які компілюються в інтерпретатор, залежатимуть від параметрів, переданих під час складання програми.

Файл \_\_init.py\_\_

Раніше в пакетах обов'язково потрібно було розмістити допоміжний файл \_\_init\_\_.py.

Файл \_\_init\_\_.py — це службовий файл, який інтерпретатор обов'язково виконає під час першого імпорту пакету.

Зазвичай \_\_init\_\_.py — порожній і нічого не робить. Але використовується коли в проєкті складна структура.

Наприклад, структура проєкту виглядатиме так:

📦example\_init

┣ 📂utility

┃ ┣ 📂dummy

┃ ┃ ┗ 📜functions.py *# not\_bad*

┃ ┣ 📂useful

┃ ┃ ┗ 📜functions.py *# nice\_function*

┃ ┗ 📜\_\_init\_\_.py

┗ 📜main.py

Якщо залишити файл \_\_init\_\_.py порожнім, то використання функцій nice\_function та not\_bad буде виглядати якось так:

import utility

utility.useful.functions.nice\_function()

utility.dummy.functions.not\_bad("Test string")

Якщо ж розробник подумав про користувача то \_\_init\_\_.py має виглядає ось так:

from utility.useful.functions import nice\_function

from utility.dummy.functions import not\_bad

\_\_all\_\_ = ['nice\_function', 'not\_bad']

Зверніть увагу на константу \_\_all\_\_ — це список модулів або пакетів, які імпортуються, якщо у виразі from ... import \* в кінці вказаний символ \*.

Тепер можна скористатися таким імпортом функцій з пакету:

from utility import nice\_function, not\_bad # або from utility import \*

nice\_function()

not\_bad("Test string")

Пакетний менеджер pip та створення віртуального оточення

Пакетний менеджер **pip** дозволяє встановлювати, оновлювати та видаляти бібліотеки та інструменти, що використовуються в програмуванні. Повна документація з **pip** доступна на [сторінці офіційної документації](https://pip.pypa.io/en/stable/user_guide/).

Вивести список встановлених пакетів:

python -m pip list

Часто в системі прописано, що pip запускається за допомогою Python і ви можете викликати pip напряму:

pip list

Для встановлення останньої версії пакету, на прикладі пакету **requests**:

 pip install requests

Для встановлення конкретної версії пакету **requests**:

pip install requests==2.28.2

Встановлення версії пакету **requests** новішого за 2.28.2:

pip install requests>=2.28.2

Встановлення версії пакету **requests** давнішого за 2.28.2:

pip install requests<=2.28.2

Видалення пакету **requests**:

pip uninstall requests

Список встановлених пакетів з версіями:

pip freeze

[Створення віртуального оточення](https://docs.python.org/3/library/venv.html)

☝ Створення віртуального оточення в Python важливе, оскільки це дозволяє ізолювати залежності для кожного проєкту. Це означає, що можна мати різні версії бібліотек для різних проєктів і таким чином уникнути конфліктів залежностей.

Щоб створити нове віртуальне оточення, виконайте наступні кроки:

1. Відкрийте термінал або командний рядок.
2. Перейдіть до директорії, де ви хочете створити свій Python проєкт.
3. Виконайте наступну команду для створення віртуального оточення:

python -m venv .venv

Це створить директорію **.venv** у поточній директорії, яка міститиме віртуальне оточення.

Активація віртуального оточення(повернутися в це віртуальне оточення виходу):

.\.venv\Scripts\activate.bat *# На Windows у командному рядку (CMD)*  
.\.venv\Scripts\Activate.ps1 *# На Windows у PowerShell; термінал в VS Сode*

source .venv/bin/activate *# На macOS*

Після активації командний рядок зміниться, відображаючи назву віртуального оточення, це показує, що воно активне.

Щоб встановити необхідні бібліотеки в межах цього віртуального оточення, використовуйте **pip install**:

pip install package\_name

Щоб повернутися до системного глобального середовища Python, виконайте в консолі:

deactivate

Щоб видалити віртуальне оточення, достатньо фізично видалити директорію **.venv** з усім її вмістом в директорії проєкту.

Файл **requirements.txt** дає список всіх встановлених пакетів у вашому віртуальному середовищі та їхні версії. Можна автоматично згенерувати **requirements.txt**:

pip freeze > requirements.txt

Щоб встановити пакети, перелічені у **requirements.txt**, використовуйте команду:

pip install -r requirements.txt

Бібліотека colorama

**Colorama** - це популярна бібліотека Python, яка використовується для додавання кольорів до тексту в консолі.

from colorama import Fore, Back, Style

print(Fore.RED + 'Це червоний текст')

print(Back.GREEN + 'Це текст на зеленому фоні')

print(Style.RESET\_ALL) *# скидання стилю до стандартного; Fore.RESET*

print('Це звичайний текст після скидання стилю')

Єдина точка входу проєкту

Щоб об'єднати файли в проєкті, можна імпортувати функції або класи з інших файлів у main.py:

from my\_module import my\_function

def main():

my\_function()

if \_\_name\_\_ == "\_\_main\_\_":

main()

Це необхідно робити, щоб було простіше змінювати різні частини коду, наприклад змініти щось в одній функції, і щоб довго не шукати її розташування, її можна помістити в інший відповідний файл.

Python Core 2.0 ֍ 5) Функціональне програмування та вбудовані модулі Python

Можливості деяких вбудованих модулів Python

Модуль collections

Модуль **collections** в Python - це вбудований модуль, який містить спеціалізовані типи даних, які надають альтернативи стандартним вбудованим контейнерам. Іменовані кортежі

Іменовані кортежі **namedtuple** – це розширення стандартного типу даних **tuple**, яке дозволяє виконувати доступ до елементів списку за іменем, а не за індексом.

Point = collections.namedtuple('Point', ['x', 'y']) *# Створення іменованого кортежу*

Cat = collections.namedtuple('Cat', ['nickname', 'age', 'owner'])

cat = Cat('Simon', 4, 'Krabat')

print(f'This is a cat {cat.nickname}, {cat.age} age, his owner {cat.owner}')

Виведення:

This is a cat Simon, 4 age, his owner Krabat

Counter

✂️ Цей код можна запустити!

**Counter** належить до модуля **collections** і служить для підрахунку гешабельних об'єктів. **Counter** функціонує як словник, де ключами є елементи, а значеннями - їх кількість у колекції.

mark\_counts = collections.Counter(marks) *# можна передати текстовий рядок*

Один з найпопулярніших методів **Counter** - це **most\_common()**, який повертає список елементів та їх частоту, починаючи з тих які зустрічаються найчастіше.

defaultdict

У звичайному словнику Python спроба доступу до ключа, якого не існує, викликає виняток **KeyError**. В **defaultdict**, якщо ключ не існує, він автоматично створюється зі значенням, яке повертає функція, передана під час створення **defaultdict**.

Під час створення **defaultdict**, ви повинні передати функцію, яка повертає значення за замовчуванням для нових ключів. словник[нове\_значення].append(значення\_за\_замовчуванням)

Структури даних: Стек, Черга та Двостороння черга

Стек

Стек - структура даних, яка дозволяє здійснювати операції вставки і вилучення даних за принципом "**Останнім прийшов - першим вийшов**" (LIFO - Last In, First Out).

def create\_stack(): *# Створення стеку*

return []

def is\_empty(stack): *# Перевірка на порожнечу*

return len(stack) == 0

def push(stack, item): *# Додавання елементу*

stack.append(item)

def pop(stack): *# Вилучення елементу*

if not is\_empty(stack):

return stack.pop()

else:

print("Стек порожній")

def peek(stack): *# Перегляд верхнього елемента*

if not is\_empty(stack):

return stack[-1]

else:

print("Стек порожній")

Черга

Черга (queue) – це абстрактна структура даних, яка діє за принципом "**першим прийшов – першим вийшов**" (FIFO: First In, First Out).

Використання **deque** з модуля **collections** в якості черги.

from collections import deque

queue = deque() *# Створення черги*

queue.append(…) *# Enqueue: Додавання елемента*

print("Черга після додавання елементів:", list(queue))

print("Видалений елемент:", queue.popleft()) *# Dequeue: Видалення елемента*

print("Перший елемент у черзі:", queue[0])  *# Peek: Перегляд першого елемента*

print("Чи черга порожня:", len(queue) == 0) *# IsEmpty: Перевірка на порожнечу*

print("Розмір черги:", len(queue)) *# Size: Розмір черги*

Двостороння черга deque

Двостороння черга, або Deque (скорочення від "double-ended queue"), є типом структури даних, яка дозволяє вставляти та видаляти елементи з обох кінців.

✂️ Цей код можна запустити!

from collections import deque

d = deque() *# Створення пустої двосторонньої черги*

d.append('middle') *# Додаємо елемент в кінець черги*

d.appendleft('first') *# Додаємо елемент на початок черги*

print(list(d)) *# Виведення поточного стану черги*

print(d.pop()) *# Видалення та виведення останнього елемента (з правого кінця)*

print(d.popleft()) *# Видалення та виведення першого елемента (з лівого кінця)*

Ще однією особливістю deque є можливість, (нові елементи витісняють старіші, але розмір залишається незмінним):

d = deque(maxlen=5)

Контроль точності обчислень decimal

**Decimal** — це клас у модулі **decimal**, який забезпечує точну арифметику з дійсними числами, вирішуючи деякі проблеми, які виникають при використанні типу **float**.

✂️ Цей код можна запустити!

from decimal import Decimal

Об'єкти Decimal не можна використовувати в одному виразі разом з float.

from decimal import getcontext

getcontext().prec = 4

Тепер будь-які обчислення з **Decimal** будуть мати точність до 4 знаків значуших цифр.

Округлити число до двох знаків після коми, ви використовуєте **Decimal** об'єкт з двома нулями після коми як шаблон.

from decimal import Decimal, ROUND\_DOWN

# Вихідне число Decimal

number = Decimal('3.14159')

# Встановлення точності до двох знаків після коми

rounded\_number = number.**quantize(**Decimal('0.00'), rounding=ROUND\_DOWN**)**

**Decimal** дозволяє вибирати різні режими округлення:

* ROUND\_FLOOR число завжди округляє до найближчого меншого значення.
* ROUND\_CEILING число завжди округляє до найближчого більшого значення.
* ROUND\_HALF\_DOWN числа округлюються до найближчого значення. У випадку, коли число знаходиться точно посередині між двома можливими варіантами число округляється вниз, тобто до найближчого меншого значення.
* ROUND\_HALF\_UP числа округлюються до найближчого значення. Проте у випадку нічиї, число округляється вгору, тобто до найближчого більшого значення.
* ROUND\_UP число округляється від нуля. Це означає, що додатні числа округлюються до більшого, а від'ємні - до більшого за модулем значення.
* ROUND\_DOWN число округляється до нуля. Тобто додатні числа округлюються до меншого, а від'ємні - до меншого за модулем значення.
* ROUND\_HALF\_EVEN числа округлюються до найближчого числа. Цей режим, також відомий як "банківське округлення", округлює число до найближчого значення, але у випадку нічиї, воно округляється до найближчого парного цілого числа. **(За замовчуванням)**

Генератори

Генератор дозволяє нам оброблювати дуже великі файли і при цьому економити пам'ять.

Один із способів створити генератор у Python — це створити особливу функцію з декількома точками входу. Для цього використовується ключове слово **yield**.

Оператор **yield** при наступному зверненні не розпочинає виконання функції з початку, а продовжує з місця зупинки функції.

def my\_generator():

yield 1

yield 2

gen = my\_generator()

print(next(gen)) *# Виведе 1*

print(next(gen)) *# Виведе 2 # Продовжує з місця зупинки*

Після того, як генератор повернув усі свої значення, якщо ми виконаємо виклик **next(gen),** то виникне виняток **StopIteration**. Для контролю винятку **StopIteration** найчастіше генератори використовуються в циклах for ...:

def count\_down(start):

while start > 0:

yield start

start -= 1

for number in count\_down(5):

print(number)

Елементи функціонального програмування

Функція як об'єкт першого класу

Функції в мові програмування використовуються як звичайні об'єкти.

Перше, це присвоїмо функцію змінній.

def my\_function():

print("Hello, world!")

f = my\_function

f() *# Виведення: Hello, world!*

Функції можуть бути аргументами інших функцій.

from typing import Callable

def add(a: int, b: int) -> int:

return a + b

def multiply(a: int, b: int) -> int:

return a \* b

def apply\_operation(a: int, b: int, **operation: Callable[[int, int], int]**) -> int:

return operation(a, b)

result\_add = apply\_operation(5, 3, add) *# Використання*

result\_multiply = apply\_operation(5, 3, multiply)

Функції як об'єкт першого класу можуть повертають інші функції.

from typing import Callable

def power(exponent: int) -> Callable[[int], int]:

def inner(base: int) -> int:

return base \*\* exponent

return inner

square = power(2) *# Використання*

cube = power(3)

print(square(4)) *# 16*

print(cube(4)) *# 64*

І останнє: це зберігання функцій у структурах даних.

*# Словник операцій (функції, які ми прописали до цього)*

operations: Dict[str, Callable] = {

'add': add,

'multiply': multiply,

'square': square,

'cube': cube}

result\_add = operations['add'](10, 20) *# 30*

result\_square = operations['square'](5) *# 25*

Замикання та каррування

[**Замикання**](https://uk.wikipedia.org/wiki/%D0%97%D0%B0%D0%BC%D0%B8%D0%BA%D0%B0%D0%BD%D0%BD%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)) (closure) відбувається, коли внутрішня функція запам'ятовує стан свого оточення в момент свого створення і може використовувати ці змінні навіть після того, як зовнішня функція завершила своє виконання.

Ключові аспекти замикань:

1. Внутрішня функція має доступ до змінних, визначених у області видимості зовнішньої функції.
2. Зовнішня функція повертає внутрішню функцію як результат своєї роботи.
3. Після завершення роботи зовнішньої функції, внутрішня функція зберігає доступ до цих змінних, що відіграє важливу роль у певних програмних патернах та алгоритмах.

[**Каррінг**](https://uk.wikipedia.org/wiki/%D0%9A%D0%B0%D1%80%D1%80%D1%96%D0%BD%D0%B3_(%D1%96%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) (currying) — це техніка в програмуванні, коли функція, яка приймає кілька аргументів, перетворюється на послідовність функцій, кожна з яких приймає один аргумент.

Декоратори

Декоратори дозволяють змінювати поведінку функцій або методів без зміни їхнього вихідного коду. Вони є прикладом функцій вищого порядку, які приймають іншу функцію як аргумент та повертають нову функцію.

Декоратори використовуються з синтаксисом **@**, що робить їх застосування простим та елегантним.

Дуже важливо при створенні декораторів використовувати модуль **functools**, це необхідно для збереження метаданих оригінальної функції, яку ми декоруємо. Функція **functools.wraps** допомагає в цьому, зберігаючи інформацію про оригінальну функцію, як-от ім'я функції та документацію.

from functools import wraps

def logger(func):

**@wraps(func)**

def inner(x: int, y: int) -> int:

print(f"Викликається функція: {func.\_\_name\_\_}: {x}, {y}")

result = func(x, y)

print(f"Функція {func.\_\_name\_\_} завершила виконання: {result}")

return result

return inner

**@logger**

def complicated(x: int, y: int) -> int:

return x + y

print(complicated(2, 3)) *# 5*

print(complicated.\_\_name\_\_) *# complicated*

Викликається функція: complicated: 2, 3

Функція complicated завершила виконання: 5

Comprehensions

В програмуванні досить часто необхідно створити заповнену колекцію. Для спрощення таких операцій в Python ввели конструкції **Comprehensions**.

**Comprehensions** в Python - це спосіб компактного створення колекцій на основі наявних колекцій.

* List Comprehensions

List comprehensions використовуються для створення нових списків та мають наступний синтаксис:

[new\_item for item in iterable if condition]

* Set Comprehensions

Set comprehensions використовуються аналогічно list comprehensions, але для створення множин. Синтаксис:

{new\_item for item in iterable if condition}

* Dictionary Comprehensions

Dictionary comprehensions використовуються для створення нових словників. Синтаксис:

{key: value for item in iterable if condition}

Функціональне програмування в Python

Лямбда-функції

Лямбда-функції використовуються для створення маленьких, однорядкових функцій.

Синтаксис лямбда-функції є наступним:

Ім’я\_функції = lambda arguments: expression

print(Ім’я\_функції(параметри))

Насправді це ***поганий тон*** зберігати лямбда-функції у змінних, вони повинні створюватися там, де будуть використовуватися і більше ніде не залишати слідів.

print((lambda arguments: expression)(параметри))

Лямбда-функції часто використовуються як аргументи для функцій вищого порядку, таких як **map()**, **filter()** або **sorted()**.

nums\_sorted = sorted(список, key=lambda x: -x) *# зворотнє сортування*

Функція map

Один з прикладів використання лямбда-функцій — це генератор map.

Функція **map()** є функцією вищого порядку, що означає, що вона приймає іншу функцію як аргумент. Синтаксис:

map(function, iterable, ...)

* **function -** функція, яку треба застосувати до кожного елемента в **iterable**.
* **iterable -** об'єкт ітерації (список, кортеж тощо), елементи якого будуть оброблятися **function**.

Можна застосувати map до декількох списків

Але після того як у Python з'явився **list comprehensions** його використовують для того ж самого призначення, що забезпечує більшу читабельність та виразність. Для двох списків ми теж можемо використати **list comprehensions** за допомогою функції **zip**

Функція filter

Функція **filter()** використовується для фільтрації об'єктів ітерації, таких як списки або кортежі, за допомогою заданої функції. Синтаксис **filter():**

filter(function, iterable)

* **function** - функція, яка визначає, чи слід включати елемент у результат. Функція повинна приймати один аргумент і повертати булеве значення **True** або **False**.
* **iterable** - об'єкт ітерації (наприклад, список, кортеж), елементи якого будуть перевірятися функцією **function**.

Не обов'язково використовувати lambda функцію.

Хоча **filter()** може бути корисним для багатьох сценаріїв, у деяких випадках **list comprehensions** можуть забезпечити більш читабельний та ефективний спосіб досягнення тих самих цілей.

Функція any

Функція **any()** є вбудованою функцією, яка повертає **True**, якщо хоча б один елемент із заданого об'єкта ітерації є істинним. Якщо об'єкт ітерації порожній або всі його елементи є хибними, то **any()** повертає **False**. Основне застосування функції any у сценаріях, де потрібно визначити, чи хоча б один елемент в колекції відповідає певній умові. Досить часто її використовують для поєднання з іншими функціями, такими як **map()** або list comprehensions, для більш складних умов перевірки.

Функція all

Функція **all()** є вбудованою функцією, яка повертає **True**, якщо всі елементи в переданому їй об'єкті ітерації є істинними - тобто не **False**, **0**, **""**, **None** або будь-яке інше значення, яке Python оцінює як хибне. Але будьте уважні, якщо об'єкт ітерації порожній, функція **all()** повертає **True**.

Python Core 2.0 ֍ 6) Основна робота з класами

Python Core 2.0 ֍ 7) Розширене Об'єктно-Орієнтоване Програмування в Python

Python Core 2.0 ֍ 8) Серіалізація та копіювання об'єктів в Python